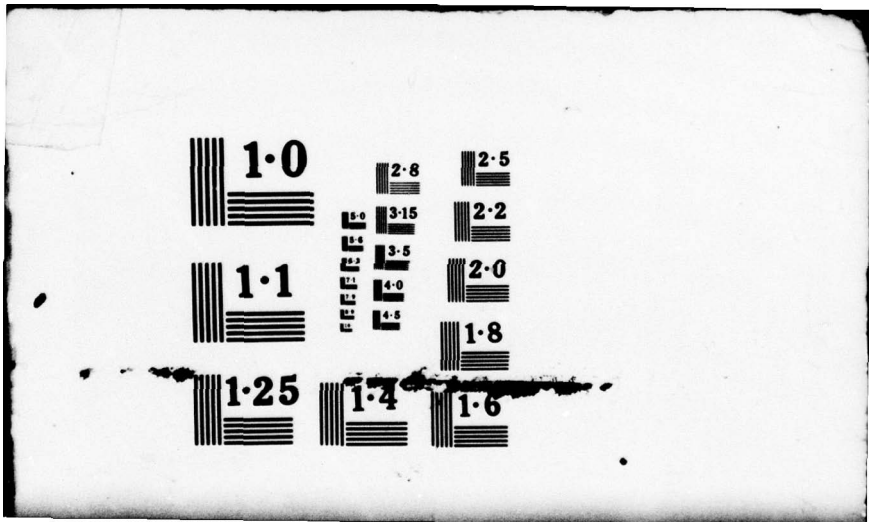
AD-A066 142 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS F/G 9/2
INTERLISP-11.(U)
MAR 79 A K HARTLEY N00014-77-C-0480
UNCLASSIFIED BBN-4076 NL

1 OF 1
ADA
066142

END
DATE
FILMED
5 79
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5

Bolt Beranek and Newman Inc.

LEVEL II

12

Report No. 4076

Interlisp-11

Final Technical Report
1 July 1977 to 30 June 1978

March 1979

Prepared for:
Defense Advanced Research Projects Agency

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: BBN [redacted]-4076	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): INTERLISP-11.	5. TYPE OF REPORT & PERIOD COVERED: FINAL TECHNICAL REPORT. 1-Jul-77 to 31-Jun-78
	6. PERFORMING ORG. REPORT NUMBER: BBN Report No. 4076
7. AUTHOR(s): Alice K. Hartley	8. CONTRACT OR GRANT NUMBER(s): N0014-77-C-0480
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge MA 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 6D30
11. CONTROLLING OFFICE NAME AND ADDRESS: ONR Department of the Navy Arlington VA 22217	12. REPORT DATE: Mar 79
	13. NUMBER OF PAGES: 43 P.
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Dept. of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

LISP, Interlisp, LISP machine, microcoding

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the implementation of an Interlisp system on a PDP-11/40 with writable control store and an extended virtual address space. The system provides on a minicomputer the capability and performance previously available only on large time sharing systems.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

060 100

BBN Report No. 4076 March 1979

Interlisp-11

Final Technical Report
July 1, 1977 to June 30, 1978

Alice K. Hartley

Prepared for:

Advanced Research Projects Agency
ARPA Order No. 3415

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by ONR under Contract No. N00014-77-C-0480.

TABLE OF CONTENTS

1. INTRODUCTION

The work described in this report is a continuation of work performed under Contract No. N00014-76-0476 to develop an Intelligent On-Line Assistant and Tutor system. As part of that contract we began the development of an Interlisp system for the PDP-11/40 that we call Interlisp-11. At the end of that contract we had an Interlisp-11 facility that could run LISP code compiled elsewhere, and a running interpreter. The goals of this contract were to incorporate the rest of the Interlisp programming environment, provide the Interlisp spaghetti stack facility [3], implement a garbage collector, provide user data types, develop a local file system, and provide display software for windowing, scrolling, etc. For the purpose of having a complete description of the current Interlisp-11 system we have included here descriptions of parts of the system already reported in [1].

The goals for the implementation of Interlisp-11 were:

- single user system
- fully compatible with Interlisp-10
- performance approximately one half that of the then current KA10 processor
- maximize machine independence for easy transfer to new hardware
- provide larger virtual address space than the 256K 36 bit words of the PDP-10.

2. HARDWARE

The hardware components of the system are:

- PDP-11/40 processor
- microcode writable control store (WCS) designed by Fuller et al [8] and built by Three Rivers Computer Corp.
- BBN memory map
- 128K words of memory
- 56 MB disk
- real time clock
- alpha numeric terminal
- bit map display terminal
- 9600 baud interface to a TOPS-20 system for file transfer.

We also have an IMP-11 interface for connection to the ARPANET and an RS04 fixed head disk that can be used for swapping. However, neither of these components is required by the software. They are currently used only for bootstrapping.

Below we describe the characteristics of the two unusual hardware components, the WCS and the BBN memory map.

The required or desirable features of the microcode for a LISP machine, or for any other language emulator, are:

- A large amount of microcode storage. It is difficult to give a precise estimate because the amount of space required depends on the structure of the microcode, but on the order of 4K words is probably sufficient if the microcode is at all reasonable. Of course, more is always better.
- The ability to quickly and easily dispatch on arbitrary contiguous bit fields is required for instruction and data decoding.
- The ability to generate constants from the microcode for masks, arithmetic, trap vector locations, etc. A less appealing solution is to provide enough registers to contain all the required constants.
- Data paths and ALU should be at least as wide as the virtual address size.

- Large number of internal registers.
- Stack for microcode subroutine calls.

The other requirement for LISP is a large virtual address space, with the ability to perform demand paging. Specifically the requirements are:

- Virtual address space of 4-16 million 16 bit words.
- Page size small enough. The optimum page size for LISP is unknown, but we believe that pages 4K words or larger are clearly too big. (Some manufacturers think 64K words is a page!)
- Enough status information to determine which page caused a page fault and to allow resumption of execution after the page fault has been serviced.

The 11/40E (11/40 with WCS) has 1024 80 bit words of writable microprogram control storage in addition to the 256 56 bit words of read-only storage in the basic PDP-11/40 processor. The 11/40 has 16 16 bit internal registers, 16 bit data paths, and one 16 bit ALU. The basic 11/40 microprocessor was designed specifically to implement the PDP-11 instruction set and is not at all suited for general emulation. The WCS was designed with general emulation in mind and therefore augments the basic microprocessor capabilities and incorporates most of the desirable features mentioned above.

Specifically the WCS has a general purpose mask-shift unit to extract arbitrary fields for branch control in the microprogram and to assist the basic ALU in the manipulation of data. There is a 16 bit literal field for the generation of masks and constants. There is a microsubroutine facility that uses a 16 word by 16 bit stack for temporary values and microsubroutine return addresses. The WCS also has convenient carry control for multiple precision arithmetic.

While the 11/40E processor meets most of the criteria for the microcode, it has some limitations. The amount of control store is too small, there are too few registers, and the data paths are not wide enough. Since we were aiming for modest performance, we decided that the limitations would not be fatal especially because the native PDP-11 instruction set can be used for those operations that will not fit in microcode; the fact that the microcode stack can hold data as well as microsubroutine return addresses compensates somewhat for the small number of registers; and the need for double precision address arithmetic can be avoided by imposing some restrictions on data allocation.

The PDP-11/40E does not meet the virtual address space requirements at all. It has a user address space of 32K words with 4K word pages. To provide a large virtual address space we built a simple memory map, a linear page table mapping a virtual page to a real

memory page. The PDP-11 uses byte addressing which is not particularly well suited to LISP since in most cases the smallest unit we want to address is a 16 bit word and byte addressing would waste one bit in every pointer. For this reason, memory references in LISP mode use word addressing. The decisions to be made were the size of the virtual address space and the page size. A 22 bit word address with a page size of 1K words requires a 4K entry page table. In spite of the fact that a larger address space would be desirable, and a smaller page size might be better, the fact that a 4K page table can be built compactly using 4K ram chips (the largest available 50nsec chip), led to the above parameters for the first version. The components of the map are:

- A 6 bit register, EXT, that contains the high order 6 bits of a 22 bit virtual address. EXT can be written from the microcode using a previously unused value in one of the microcode fields. EXT is write only.

- The 4K by 10 bit page table. Each entry in the page table has 8 bits for the real memory page (only 7 bits are used currently), 1 bit for page valid/invalid, and 1 bit for page writable/not writable.

- 3 mode bits in the high order bits of the program status register to control addressing modes as follows:

  000 Kernel mode; PDP-11 mode addressing to the second half of segment 0; all instructions are legal.

  100 User mode; PDP-11 mode addressing to the first half of segment 0; the PDP-11 instructions HALT and RESET will trap.

  101 User mode; LISP mode addressing using 22 bit word addresses; differs from mode 111 below only in that after a page fault, instruction execution must resume in PDP-11 mode; used when executing the PDP-11 mode extended operations.

  111 User mode; LISP mode addressing using 22 bit word addresses; LISP mode exectuion.

  We use 3 bits rather than 2 here for programming convenience.

- Various status registers.

- A 256 x 2 bit PROM, similar to the PROM in the DEC memory management unit, to control the basic PDP-11 microcode for traps and interrupts. It forces addressing in the new program status mode when storing the old program counter and program status on the (new mode) stack.

3. SOFTWARE

The Interlisp-11 implementation has four components: the operating environment, the LISP kernel, the L-code instruction set including the microcode to interpret it and the compiler to generate it, and the Interlisp system including LISP functions and data.

The operating environment provides the facilities of a primitive operating system. These include page management, interfaces to I/O devices, handling of traps and interrupts, and the basis for a file system. The operating environment is written in native PDP-11 code with the addition of a few new instructions implemented in microcode. In particular, instructions to save and restore the LISP environment are required because some of the general registers used by LISP are not accessible to PDP-11 code (only 8 of the 16 general registers are visible to the native instruction set). Other instructions allow reading and writing locations in the LISP virtual address space. The operating environment is called from the LISP kernel using PDP-11 EMTs. It passes some traps from the microcode (e.g. stack overflow, non-numeric argument) to the LISP kernel using a table of trap vectors at a fixed location in the LISP kernel.

The entire file directory and file lookup are implemented in LISP. We describe here the file system primitives that are provided by the operating environment. There are a fixed number of permissible files, currently 1024. At the lowest level files are referenced by number. The operating environment keeps an up to date directory on the disk of the addresses of the index blocks of the files. When a file is opened it is assigned an open file number OFN. One segment of the virtual address space is dedicated to file page buffers and the OFN essentially provides the address of a page in the file segment (the address can be computed from the OFN). The index block of each existing file contains, in addition to the disk addresses of the pages of the file, space for other useful information such as the length of the file in bytes, the name of the file (to permit reconstruction of the file directory), and potentially dates of creation and last reference. The primitives are:

OPEN(N) - open file number N, value is an OFN.

CLOSE(OFN) - close the file specified by OFN.

DELETE(N) - delete file number N. File must not be open.

PMAP(OFN N) - map the Nth page of the file specified by OFN to the file segment, unmapping any page that might be there.

GETIB(N M) - value is the contents of the Mth word in the index block of file number N. Also returns in a second register, information as to whether or not N references an existing file or is beyond the range of possible file numbers.

SETIB(N M V)- set the Mth word in the index block of the file number N to V.

GETNFX() - assign and return the next available file number.

SYSOUT(N) - save the contents of the Interlisp virtual memory on file number N.

SYSIN(N) - run the Interlisp image on file number N.

The LISP kernel provides initialization, storage allocation, atom hashing, garbage collection, stack searching primitives, parts of the interpreter, integer multiply and divide, floating point subroutines, and in general a lot of primitives that should have been implemented in microcode but wouldn't fit. It also contains some hand written L-code for things that could have been written in LISP but are required in order to load compiled code. The Interlisp READ and PRINT functions are not part of the LISP kernel, but some rudimentary substitutes exist for bootstrapping purposes. The kernel is implemented in a combination of PDP-11 code and L-code. (We have defined a new PDP-11 instruction to enter LISP mode, and an L-code instruction to leave LISP mode.) The LISP kernel calls the operating environment using PDP-11 EMTs with the arguments in registers. It calls LISP functions using the SCALL instruction (described below) requiring that the names of the functions be in the system constant table. The LISP kernel calls itself using the miscellaneous opcodes and is also called from Interlisp by means of miscellaneous opcodes which in effect do a PUSHJ. There are no SUBRs.

Interlisp-11 provides a new data type, an IO descriptor, which contains information about open files or IO devices. The most important components of an IO descriptor are a buffer pointer and the name of a function to call when the buffer is full or empty. An IO descriptor can be set up with an empty buffer, with the effect that a function will be called for each character written or read. This gives the user total control over Interlisp IO. This facility was used to interface the bit map terminal to Interlisp-11 and simplified the implementation of echoing, scrolling, and windowing.

Instruction Set

The L-code instructions are 16 bits wide with a 6, 7, or bit opcode and a 10, 9 or 8 bit operand. The operand specification usually consists of a few bits for the operand type and an offset from a base implied by the type.

The operand bases are:

vp - A register containing the location of the beginning of the current basic frame in the stack segment.

pp - A register containing the location of the end of the current frame extension in the stack segment.

LIT - The location of the literal area of the current running function. LIT is a 22 bit pointer contained in 1 1/2 registers. The literal area includes the function entry frame (FEF) for the function which carries information about which variables are SPECVARs; access information for local variables bound in outer frames; 16 bit pointers to value calls of SPECVARS referenced by the function; and pointers to constants referenced by the function.

SYSTEM CONSTANT - The base address of a table containing frequently used constants such as T and NIL is a fixed memory location.

IMMEDIATE - Immediate operands are the integers in the range - 64 to + 63. The operand is the sign extended offset.

For those instructions that have a 10 bit source operand, the source type is 2 or 3 bits and the offset is 7 or 8 bits.

The source types are:

STACK - Source type 000, 7 bit offset. References a temporary value in the current frame extension. The offset is a positive quantity giving the number of stack slots (a stack slot is 2 16 bit words) back from the end of the frame extension (PP).

LOCAL - Source type 001, 7 bit offset. References a local variable, i.e. a non SPECVAR, in the current basic frame. The offset is a positive quantity specifying the number of stack slots from the beginning of the basic frame (VP).

SPEC - Source type 010, 7 bit offset. References a 16 bit pointer to the value cell of a SPECVAR. If a function is compiled, the value cell pointers are in the literal area of the function. The offset is a postive quantity specifying the number of words from the beginning of the literal area (LIT).

PVAR - Source type 011, 7 bit offset. Used by PROGs and open LAMBDAs to reference local variables bound in outer frames. The base for the reference is the literal pointer (LIT) and the offset is the number of words from the beginning of the literal area. A PVAR literal is a 16 bit quantity containing the number of frames back, and the desired argument number.

LIT - Source type 10X, 8 bit offset. References a literal of the currently running function. Literals are 22 0r 32 bit quantities occupying 2 words. The offset is the number of 2 word quantities from the beginning of the literal area.

SYSTEM CONSTANT - Source type 110, 7 bit offset. Constants that are referenced frequently, such a T and NIL, as well as constants required by the handwritten L-code, such as LAMBDA and NLAMBDA, are

contained in the system constant table at a fixed location in memory. Each constant occupies 2 words and the offset is the number of 2 word entries from the beginning of the table.

IMMEDIATE - Source type 111, 7 bit offset. Immediate operands are the integers -64 thru +63. The integer is the sign extended offset.

The 6 bit opcodes with 10 bit source operand as described above are:

040XXX	PUSH	Pushes the operand on the stack.
042XXX	RET	Return from current function with operand as value.
044XXX	unused	
046XXX	unused	
050XXX	CAR	Push CAR of the operand on the stack.
052XXX	RCAR	Return from current function with CAR of the operand as value.
054XXX	CDR	Push CDR of the operand on the stack.
056XXX	RCDR	Return from current function with CDR of the operand as value.
060XXX	CAAR	Push CAAR of the operand on the stack.
062XXX	RCAAR	Return from the current function with CAAR of the operand as value.
064XXX	CADR	Push CADR of the operand on the stack.
066XXX	RCADR	Return from the current function with CADR of the operand as value.
070XXX	CDAR	Push CDAR of the operand on the stack.
072XXX	RCDAR	Return from current function with CDAR of the operand as value.
074XXX	CDDR	Push CDDR of the operand on the stack.
076XXX	RCDDR	Return from current function with CDDR of the operand as value.
020XXX	LISTP	If the operand is a list, set the indicator TRUE; else set the indicator FALSE.

022XXX	ATOM	If the operand is a litatom, integer, or floating number, set the indicator TRUE; else set the indicator false.
024XXX	LITATOM	If the operand is a litatom, set the indicator TRUE; else set the indicator FALSE.
026XXX	NUMBERP	If the operand is an integer or floating number set the indicator TRUE; else set the indicator FALSE.
030XXX	FIXP	If the operand is an integer, set the indicator TRUE; else set the indicator FALSE.
032XXX	STRINGP	If the operand is a string, set the indicator TRUE; else set the indicator FALSE.
034XXX	ARRAYP	If the operand is an array, set the indicator TRUE; else set the indicator FALSE.
036XXX	STACKP	If the operand is a stack pointer, set the indicator TRUE; else set the indicator FALSE.
100XXX	IADD	Integer sum of the operand and the top of the stack is stored on the top of the stack.
102XXX	ISUB	Integer difference of the operand and the top of the stack is stored on the top of the stack (E-TOS->TOS).
104XXX	IMUL	Integer product of the operand and the top of the stack is stored on the top of the stack.
106XXX	IDIV	Integer quotient of the operand and the top of the stack is stored on the top of the stack (E/TOS->TOS).
110XXX	IREM	Integer remainder of the operand and the top of the stack is stored on the top of stack.
112XXX	EQ	Compare the operand with the pointer on the top of the stack; set the indicator TRUE if the two are EQ; pop the stack.
114XXX	IGT	If the integer operand is greater than the integer on the top of the stack, set the indicator TRUE. Pop the stack. Error if either argument is not an integer.
116XXX	ILS	If the integer operand is less than the integer on the top of the stack, set the

		indicator TRUE. Pop the stack. Causes an error if either argument is not an integer.
120XXX	FADD	Floating point sum of the operand and the top of the stack is stored on the top of the stack.
192XXX	FSUB	Floating point difference of the operand and the top of the stack is stored on the top of the stack.
124XXX	FMUL	Floating point product of the operand and the top of the stack is stored on the top of the stack.
126XXX	FDIV	Floating point quotient of the operand and the top of the stack is stored on the top of stack.
130XXX	FREM	Floating point remainder of the operand and the top of the stack is stored on the top of the stack.
132XXX	EQP	Compare the operand with the top of the stack and pop the top of the stack. Set the indicator TRUE if the pointers are EQ, or both are integers with the same value, or both are floating point numbers with the same value, or both are stack pointers referencing the same stack frame.
134XXX	FGTP	Compare the floating point operand with the floating point number on the top of the stack and pop the stack. Set the indicator TRUE if the operand is greater. Causes an error if either argument is not a floating point number.
136XXX	FLSP	If the floating point operand is less than the floating point number on the top of the stack, set the indicator TRUE. Pop the top of the stack. Causes an error if either argument is not a floating point number.

The following instructions are the ones that store values. Since it doesn't make any sense to store to a constant, there are fewer possibilities for destinations than for sources. Thus the destination is 9 bits and the opcode is 7 bits. The destinations are:

STACK - ØØ	7 bit offset
LOCAL - Ø1	7 bit offset
SPEC - 1Ø	7 bit offset
PVAR - 11	7 bit offset

The opcodes with 9 bit operands are:

Ø1ØXXX	SCDR	Stores CDR of the operand in itself.
Ø11XXX	POP	Pops the stack to the destination. The effective address is computed after the pop.
Ø12XXX	SCDDR	Stores CDDR of the operand in itself
Ø13XXX	TOS	Stores the top of the stack (without popping) in the destination.
Ø14XXX	ADD1	Stores the operand plus 1 in itself.
Ø15XXX	PCAR	Pops the stack, takes CAR of it, and stores the result in the destination. The effective address is computed after the pop.
Ø16XXX	SUB1	Stores the operand minus 1 in itself.
Ø17XXX	PCDR	Pops the stack, takes CDR of it, and stores the result in the destination. The effective address is computed after the pop.

The branches have a 6 bit opcode and a 1Ø bit offset. The sign extended offset plus the current location is the destination of the branch. If the offset is Ø, the following word contains a 16 bit offset. The branch instructions are:

14ØXXX	BRT	Branch if the indicator is true.
142XXX	BRF	Branch if the indicator is false.
144XXX	PBNIL	If the top of the stack is NIL, pop the stack and branch; otherwise don't pop and don't branch.

146XXX	BRA	Unconditional branch.
150XXX	BNIL	Branch if the top of the stack contains NIL and pop always.
152XXX	BNN	Branch if the top of the stack is not NIL and pop always.
154XXX	NBNIL	If the top of the stack is NIL, branch but don't pop the stack; otherwise don't branch and do pop.
156XXX	NBNN	If the top of the stack is not NIL, branch but don't pop the stack; otherwise don't branch and do pop.

The following opcodes have literal or immediate operands only. The opcode is 8 bits and the offset is 8 bits.

0020XX	CALL	Call function by name or function call. The literal has the number of arguments supplied in the high 8 bits, and the function name in the remaining 24 bits.
0024XX	DCALL	Like CALL but discards the value.
0030XX	LCALL	Linked call of function. (not implemented)
0034XX	DLCALL	Linked call that discards value. (not implemented)
0040XX	PBIND	Bind operation for PROGs and open LAMBDAs that make frames. The immediate operand is the number of frames deep.
0044XX	ECALL	Call operation for ENVEVAL and ENVAPPLY. The literal operand is like a CALL literal (presumably to EVAL or APPLY). ECALL also expects the new CLINK and new ALINK on the stack in that order. Causes an illegal instruction trap if the literal operand does not reference a compiled function.
0050XX	PCALL	Call operation for PROGs and open LAMBDAs that make frames. The literal operand contains the number of arguments supplied, and the offset for the FEF pointer (a small number that gets added to the current LIT pointer).
0054XX	EQQ	Compare the top of the stack with the literal operand and set the indicator TRUE if they are EQ. (Used for SELECTQ.)

0060XX	TYPTOS	The 6 bit immediate operand is a type number. If the type of the pointer on the top of the stack is equal to the given type, set the indicator TRUE.
0062XX	DTYPTOS	Like TYPTOS but also pops the stack.
0064XX	Unused	
0070XX	FRMRUN	Run the stack frame specified in register 2. (no operand)
0074X0	FUNBIND	Unbind the Frame referenced by the unboxed stack pointer on the top of the stack. Pop the stack. (no operand)
0074X1	FBIND	Bind the frame referenced by the unboxed stack pointer on the top of the stack. Pop the stack. (no operand)

Opcodes 0-1777 take their operands (if any) on the stack and return their value (if any) on the stack. A few have an immediate operand in the low order 3 bits.

The opcodes described below are those in this group that are implemented entirely in microcode.

000010	EXCH	Exchange the top of the stack with second from the top of the stack.
000020	IBOX(X)	Integer box X.
000030	GCONS(X)	Create an empty item of type X.
000040	CONS(X Y)	Create a list cell with X as the CAR and Y as the CDR.
000050	BIND	Bind the current frame.
000060	DPOP(N)	N is a 3 bit immediate operand. Pops (and discards) N items from the stack.
00007X	IRET(N)	Return from a miscellaneous op-code of N arguments. N is a 3 bit immediate operand.
000100	BIN(X)	Inputs 1 character from a string or IO descriptor X. Skips if not empty, returning the character code. Does not skip if string or buffer is empty.

000110	BOUT(X CHAR)	Output the character code CHAR to the string or IO descriptor X. Skip if not empty, returning CHAR. Does not skip if string or buffer is full.
000120	RPLACA(L A)	Replace CAR of L with A. Value is L.
000130	RPLACD(L D)	Replace CDR of L with D. Value is L.
000140	DRPLACA(L A)	Replace CAR of L with A. Discard value.
000150	DRPLACD(L D)	Replace CDR of L with A. Discard value.
000160	IVAL()	Indicator value; push T on the stack if the indicator is True; otherwise push NIL.
000170	LLM	Leave Lisp mode (enter PDP-11 mode).
000200	R16I(P N)	Return as an integer the 16 bit quantity that is N words from the beginning of P.
000201	R16S(P N)	Return as an unboxed stack pointer the 16 bit bit quantity that is N words from the beginning of P.
000202	R16V(P N)	Return as a value cell pointer the 16 bit quantity that is N words from the beginning of P.
000210	W16(P N VAL)	Store the low order 16 bits of the pointer VAL in the location that is N words from the beginning of P.
000220	RPCAR(P N)	Return the CAR format pointer that begins N words from the beginning of P.
000230	WCAR(P N VAL)	Store the pointer VAL in CAR format in the locations that begin N words from the beginning of P.
000240	RPCDR(P N)	Return the CDR format pointer that begins N words from the beginning of P.
000250	WCDR(P N VAL)	Store the pointer VAL in CDR format in the locations that begin N words from the beginning of P.
000260	RNUM(P N)	Return as an integer the 24 bit quantity that begins N words from the beginning of P.
000270	WNUM(P N VAL)	Unbox the integer VAL and store in the location beginning N words from the

beginning of P.

00030X	SCALL(N)	Call a function. The 3 bit immediate operand N is the number of arguments supplied on the stack. Expects on the stack ARG1,...ARGN,FN, where FN may be a literal atom or function cell. If FN is not legal, eventually get to FAULTAPPLY.
000310	SCALLN(ARG1,..ARGN,FN,N)	- Same as SCALL except the number of arguments supplied is also stored on the stack.
000320	LOGOR(X Y)	Returns the logical OR of X and Y. Error if X or Y is not an integer.
000330	LOGAND(X Y)	Returns the logical AND of X and Y. Error if X or Y is not an integer.
000340	LOGXOR(X Y)	Returns the logical XOR of X and Y. Error if X or Y is not an integer.
000350	LSH(X N)	Arithmetic (sign extended) shift of X by N. Shifts left if N is positive, right if N is negative. Error if X or N is not an integer.
000360	EXFRM	Used by the interpreter to make a frame for calling an EXPR. Expects on the stack the callers PC and a number containing the size of the FEF. (The FEF is also on the stack following the arguments.)
000370	DIVAL	Indicator value. Puts T on the top of the stack (destroying previous top of stack) if the indicator is TRUE; otherwise replaces the top of the stack with NIL.

The remainder of the zero operand opcodes are implemented in L-code and/or PDP-11 code. There is a dispatch table at a fixed location in memory that contains the entry points for these opcodes (000400 - 001777). Opcodes 160000 - 177777 are unused.

Literal Atom Primitives

MKAT(STR) - Create a literal atom with pname equal to the string STR, and with the remainder of the components set to NIL or "non-existent". The value is the atom. Note: MKAT does not check for possible integer or floating point numbers.

GETPROPLIST(ATM) - Return the property list of the literal atom ATM.

SETPROPLIST(ATM VAL) - Set the property list of the literal atom ATM to be VAL. The value is VAL.

GETD(ATM) - Return the function definition of ATM. If the definition is an S-expression, the value is the S-expression (a list). If the function definition is compiled, the value is a function cell containing the definition. If ATM is not a litatom the value is NIL.

PUTD(ATM DEF) - Set the function definition of the literal atom, ATM, to be DEF. DEF may be a function cell, a literal atom, or anything else. If DEF is a literal atom, the definition of the atom is put as the definition of ATM, allowing MOVD without creation of a function cell. Anything else is put as an S-expression. If DEF is not a legitimate S-expression, an error will occur when the function is called.

FGETD(X) - If X is a literal atom, function cell, or compiled code pointer, FGETD returns the pointer portion of the function definition. If X is anything else the value is NIL. The value of FGETD can be used to test for existence of or equality (EQ) of function definitions, but will not generally be a legal argument to PUTD.

GETDP(ATM) - Returns T if ATM has a non-NIL function definition; NIL otherwise.

VCTOAT(VALUECELL) - Given a value cell pointer, returns the corresponding literal atom. (No error checks)

SET(ATM VAL) - Set the contents of the value cell of the literal atom ATM to VAL. This sets the current value of ATM (because Interlisp-11 uses shallow binding). Error if ATM not a literal atom.

AT2VC(ATM) - Return the value cell pointer of the literal atom ATM. Error if ATM is not a literal atom.

L.EVALV(ATM) - Returns the current value of the literal atom ATM or NOBIND if ATM has no value. Error if ATM is not a literal atom.

MAPATOMS(FN) - Applies FN to each literal atom that currently exists in the atom hash table.

CHARACTER(CODE) - Return the one character literal atom whose pname is the character code CODE.

List Primitives

CONS(A D) - Returns a list cell whose CAR is A and CDR is D. (implemented in micro-code)

RPLACA(L A) - Replace the CAR component of the list L by A. The value is L. (implemented in micro-code)

LIST(A1...AN N) - Takes an indefinite number of arguments, the last of which is the total number of arguments excluding itself. Returns a list of the first N arguments.

MEMB(X L) - If there is an element of the list L that is EQ to X, returns the tail of the list L beginning with X; otherwise returns NIL.

ASSOC(X L) - The value of ASSOC is the first element of L whose CAR is EQ to X. If no such element exists the value is NIL.

LAST(L) - Returns the last node in the list L. i.e. (LAST (QUOTE (A B C))) = (C) and (LAST (QUOTE (A B . C))) = (B . C). Returns NIL if L is not a list.

LENGTH(L) - Returns the number of nodes in the list L. Returns Ø if L is not a list.

NTH(L N) - Returns the tail of the list L beginning with the Nth element of L. If L has fewer than N elements, returns NIL.

Array Primitives

Arrays in Interlisp-11 conform to the Interlisp Virtual Machine Specification and therefore are different from arrays in Interlisp-1Ø.

A.ARRAY (SIZE TYP) - Allocates an array of the proper length to contain SIZE elements, where the number of words per element depends on TYP. Legal values for TYP are:

Ø - pointer; 1 - integer; 2 - floating point; 3 - hash array

The value is the array containing unboxed zeros. A.ARRAY is used to implement the more general VM function ARRAY (SIZE TYP INITVAL).

ELT(ARRAY N) - Returns the Nth element of ARRAY. If ARRAY is a hash array, returns (CONS key val). Error if ARRAY is not an array or if N is less than 1 or greater than the size of the array.

SETA(ARRAY N VAL) - Sets the Nth element of ARRAY to VAL. Error if ARRAY is not an ARRAY or if N is less that 1 or greater than the size of the array. (Note that SETA should not be used for hash arrays, but currently there is no error check.)

ARRAYSIZE(ARRAY) - Returns the number of elements in the array ARRAY. Error if ARRAY is not an array.

PUTHASH(KEY VAL ARRAY) - If ARRAY is nil use the value of SYSHASHARRAY; if ARRAY is a list, use (CAR ARRAY). If ultimate array is not a hash array, an error results. If VAL is NIL remove existing hash link from KEY in the array if any. Otherwise set the hash link of KEY to VAL in the array.

GETHASH(KEY ARRAY) - The array to use is computed as in PUTHASH. Return the hash link of KEY in the array if any. Otherwise return NIL.

CLRHASH(ARRAY) - The array to use is computed as in PUTHASH. Remove all hash links from ARRAY.

Compiled Code Primitives

In Interlisp-11 compiled code is a separate data type. (In Interlisp-10 compiled code is stored as arrays.)

A.CODE(N) - Allocates N words of compiled code where N includes the compiled code overhead. Return a pointer to the code block.

A.FNCELL(ARGTYPE NARGS DEF) - Create a function cell for a compiled code definition. ARGTYPE is the argument type of the compiled function (see ARGTYPE below), NARGS is the number of arguments the function has, and DEF is a pointer to the compiled code block.

ARGTYPE(X) - If X is a literal atom, then return the argument type of the definition of X. If X is a function cell, return the argument type of the contents of the function cell. Otherwise regard X as an S-expression. Possible values are:

0	regular LAMBDA
1	LAMBDA NO-SPREAD
2	regular NLAMBDA
3	NLAMBDA NO-SPREAD
NIL	none of the above

NARGS1(x) - Returns the number of arguments expected by the function definition of X if X is a litatom, or by the contents of X if X is a function cell. Only makes sense if the function is compiled. No error checks.

CCODEP(X) - Returns T if X is a literal atom whose function definition is compiled or if X is a function cell ditto. Otherwise returns NIL.

String Primitives

A.STPT() - Return a new string pointer referencing an empty string.

A.DSTR(N) - Allocate a dummy string; that is, create a string pointer referencing a string of N characters whose contents are unspecified.

L.CSP(OLD NEW) - Copy the contents of string pointer OLD to strong pointer NEW. Thus NEW and OLD will be equal strings, referencing the same string characters. Value is NEW. (No error checks!)

L.SBST(STR1 N M STR2) - A primitive form of the Interlisp function SUBSTRING (and used to implement SUBSTRING). Creates a substring of the string STR1, consisting of the Nth thru Mth characters of STR1. The result is smashed into the string (pointer) STR2. N must be positive, and STR2 must be a string pointer. If M is negative it means the Mth character from the end of STR1. Returns NIL if the substring is ill defined. Unlike SUBSTRING, L.SBST returns the empty string if N equals M.

ATTOSTR(ATM STR) - Smashes the string pointer STR to reference the pname of the literal atom ATM. Value is STR. Error if ATM not a literal atom or STR not a string.

SAFSTRING(STR) - Assures that the string pointer STR is "safe". String pointers can reference characters in pname space; however, it would be unhealthy to smash pnames. So SAFSTRING is used by the *function* RPSTRING to copy the characters from pname storage to string character storage if necessary.

REVOBIN(X) - Returns the character code of the last character of the string X. Removes the character from the string. Returns NIL if the string is empty (i.e. the definition of GLC is (LAMBDA (X) (CHARACTER(REVOBIN X)))).

Interpreter Primitives

L.EVAF(FORM) - Evaluates form in current context. Error if FORM is not a list. Calls (FAULTEVAL FORM) if CAR of form is not a legally defined function.

L.APPLY(FN ARGS) - Applies function to args in the current context. Calls (FAULTAPPLY FN ARGS) if FN is not a legally defined *function*.

L.APPLY*() - Implements the function APPLY* using the variables bound in the current frame. The definition of APPLY* is (LAMBDA A (L.APPLY*)).

\L.PROG(X) - Implements the function PROG. The definition of PROG is (NLAMBDA A (L.PROG A)).

L.GO(POS TAIL) - Used to implement the function GO. POS is an unboxed stack pointer to the *PROG*LAM frame of the appropriate PROG, and TAIL is the tail of the PROG including the label referenced by GO.

Basic I/O Primitives

OBIN(X) - X may be a string or IO descriptor. If X is a string, returns the character code of the first character of X and removes X from the string. If the string is empty, returns NIL. If X is an IO descriptor, returns the character code of the next character in the buffer of X. If the buffer is empty and the end function in the IO Descriptor is not NIL, applies the END function to the IO Descriptor (returning the value of the end function). If the end function is NIL returns NIL.

OBOUT(X CHAR) - X may be a string or IO descriptor. If X is a string, stores CHAR in the first character of X and removes X from the string. Returns CHAR/ if successful, or NIL if the string is empty. If X is an IO descriptor, stores CHAR in the buffer of X. Returns CHAR if successful. If the buffer is full, applies the end function of the IO descriptor (if it exists) to X and CHAR and returns the value of the end function. If the buffer is full and the end function is NIL, returns NIL.

TTYOUT(CHAR) - Output character code CHAR to the physical terminal.

TTYIN() - Get a character from the physical terminal. If none has been typed, waits till one has. Has a buffer of _ characters for type ahead and checks for terminal interrupt characters. Does not echo. Value is a character code.

L.TRATM() - Primitive RATOM used for bootstrapping purposes. Reads an atom from the physical terminal. The only delimiters recognized are space, left and right parenthesis, and line feed.

L.STRAM(IOD) - Primitive RATOM used for bootstrapping - just like L.TRATM except takes a file argument.

TTY3IN() - Get character code from direct line to system-D. Return NIL if no character available.

TTY3INWAIT() - Get character code from direct line to system-D. Wait till character is available.

TTY3INCONTROL() - Get character code from direct line to system-D. Wait till a character is available or until a control-W is typed on the terminal. Returns NIL if control-W typed.

TTY3OUT(CHAR) - Send character code on direct line to system-D.

TTY3CLR() - Clear the input buffer for the direct line to system-D. Value is NIL.

CLRBUF() - Clear the terminal input buffer.

SREADP() - Returns number of characters in the terminal input buffer if any; otherwise returns NIL.

SSYSBUF() - Sets the variable \SYSBUF to a (re-used) string containing the current contents of the terminal input buffer if the buffer is not empty. If the buffer is empty, sets \SYSBUF to NIL.

STI (CHAR) - Simulate terminal input of the character code CHAR. Value is NIL.

TVELT (N) - Return the contents (an integer) of the Nth 16 bit word in the bit map terminal memory.

TVSETA(N VAL) - Set the contents of the Nth 16 bit word in the bit map terminal memory to VAL.

L.BMTOP(N) - Used to select a variety of options in the bit map terminal.

IO Descriptor Primitives

GETIOD(X FN) - Create and return an IO descriptor. X may be a string, an integer which is assumed to be a file number, or T. If X is a string, the IODNAM field of the IOD is set to the string, and the IOD buffer is set to reference the string characters. Thus reading from a string IOD will not destroy the original string pointer. However, writing to a string IOD does clobber the string characters (presumably intentionally). If X is a file number the IODNUM field of the IOD is set to X, and the IOD buffer is set to empty. Thus the first use of the IOD will cause a call to the end function of the IOD. If X is T, the IODNAM is set to T and the buffer set to empty. The IODFN field of the IOD is set to FN. Error if X not a string, integer or T.

IODNEXTC(IOD) - Return the contents of the "next character" field of IOD. This is where READ, RATOM, etc. put the chracter that terminated the last input operation using the IOD. Value is an integer or NIL.

IODSNEXTC(IOD CHAR) - Set the "next character field" of the IOD to contain the character code CHAR. CHAR may be NIL or an integer.

IODLASTC(IOD) - Return the contents of the "last character" field of IOD. This is where READ, RATOM, etc. put the last character actually read. Value is an integer or NIL.

IODSLASTC(IOD CHAR) - Set the "last character" field of IOD to contain CHAR. CHAR may be an integer or NIL. Value is CHAR.

IODPOS(IOD) - Get the "position on line" field from IOD. Used by reading and printing functions to keep track of horizontal position.

IODSPOS(IOD N) - Set the "position on line" field of IOD to N.

IODPTR(IOD) - Return the file pointer (i.e. the number of characters read or written so far) of IOD. Causes an error if the IOD references a non-open file or if it is a purely functional IOD (neither string nor file).

IODSPTR (IOD N) - Set the file pointer of IOD to N. If N is -1 sets the file pointer to the end of the file. Causes an error if IOD is not an IO descriptor, or if N is not an integer, or if N is less than -1.

IODNUM(IOD) - Get the file number field from IOD. Value is an integer or NIL. Error if IOD is not an IO descriptor.

IODNAM(IOD) - Get the name field from IOD. The name field is either a file name, a string, T or NIL. Error if IOD not an IO descriptor.

IODFN(IOD) - Get the end function field from IOD. Error if IOD not an IO descriptor.

File System Primitives

L.OPEN(X FN FILENAME) - Opens the file specified by X. S may be a file number or an IOD. If X is an IOD, the file number field is retrieved from the IOD. If X is a file number, creates an IO descriptor for the file. Causes an error if the file does not exist. FN is stored in the function field, and FILENAME in the name field of the old or new IOD. Returns the IOD.

L.CLOSE(IOD) - Close the file referenced by IOD. Value is IOD. Error if IOD not an IO descriptor.

L.DEL (FILENUM) - Delete file whose number is FILENUM from the disk.

L.BUFF(IOD) - Get another buffer full from the file specified by IOD. Error if IOD not an IO descriptor.

RFILNM(FILENUM) - Return the name (a literal atom) in the FDB of the file specified by FILENUM.

WFILNM(FILENUM NAM) - Write the (characters of) the literal atom NAM in the FDB of the file specified by FILENUM.

GETIB(FILENUM N) - Get the contents (an integer) of the Nth word of the index block of the file specified by FILENUM. Returns NIL if

FILENUM in legal range but file non-existent. Returns T if FILENUM out of range.

SETIB(FILENUM N VAL) - Set the contents of the Nth word of the index block of the file specified by FILENUM to be VAL. Returns VAL if FILENUM is an existing file, otherwise returns T or NIL as in GETIB.

GETNFX() - Get and assign next available file number. Returns NIL if no more available.

Terminal Interrupt Primitives

GETINTCHN(CHAR) - Returns the interrupt channel number for character code CHAR if CHAR has been assigned as an interrupt character. Otherwise returns NIL.

GETINTCHAR(CHAN) - Returns the character code assigned to interrupt channel number CHAN. Returns NIL if no character assigned. Returns T if CHAN out of range.

SETINT1(CHAR CHAN HARDFLG VAR) - Assigns character code CHAR to interrupt channel CHAN. If CHAN out of range an error results. If CHAN is in user interrupt range, then, if HARDFLG is T, set to be an immediate user interrupt. If VAR is not NIL set CHAR to be a user interrupt that sets the variable VAR.

HARDP(CHAN) - Returns T if CHAN is a user interrupt channel number that is enabled for a hard interrupt. Otherwise returns NIL.

USERINTVAR(CHAN) - Returns the variable name to be set by user interrupt channel number CHAN. Return NIL if no variable or CHAN is not a user interrupt channel.

Stack Primitives

L.STKPOS(NAME N IPOS) - searches stack for the Nth occurrence of a frame named NAME beginning at IPOS. If N is negative search along C-links. If N is positive search along A-links. Error if N is not an integer. IPOS must be an unboxed stack pointer or NIL meaning start at the current frame. Returns an unboxed stack pointer to the desired frame or NIL if no such frame is found.

L.STKNTH(N IPOS) - Returns an unboxed stack pointer to the Nth frame back from IPOS. If N is negative counts back along C-links. If N is positive counts back along A-links. IPOS must be an unboxed stack pointer or NIL. Returns NIL if no such frame exists. Error if N is not an integer or if N is Ø and IPOS is NIL.

L.STACKGP(POS) - If POS is a (boxed) stack pointer, returns the unboxed contents of POS. If POS is an unboxed stack pointer, returns POS. If POS is NIL returns NIL. If POS is T returns the

topmost frame. If POS is a litatom other than NIL or T returns L.STKPOS(POS -1 NIL). If POS is an integer returns (L.STKNTH POS NIL). If POS is anything else an error occurs.

L.STKNAME(POS) - Returns the frame name of the frame at POS. POS may be either an unboxed stack pointer or NIL (the current frame).

L.STKNARGS(POS) - Returns the number of arguments in the basic frame at POS. POS may be either an unboxed stack pointer or NIL (the current frame).

L.STKARG(N POS) - Returns the value of the Nth argument in the basic frame at POS. POS may be an unboxed stack pointer or NIL.

L.STKARGNAME(N POS) - Returns the name of the Nth argument in the basic frame at POS. Returns NIL if the argument is local. POS may be an unboxed stack pointer or NIL.

L.STKSCAN(ATOM POS) - Starting at frame POS search A links for a frame in which a variable named ATOM is bound (as a SPECVAR) Returns an unboxed stack pointer to the frame if found, NIL otherwise. POS may be an unboxed stack pointer or NIL.

L.FRAMESCAN (ATOM POS) - If a variable named ATOM is bound in the stack frame at POS, returns the position of the argument (an integer) in the basic frame. Returns NIL if no such variable is bound at POS. POS may be an unboxed stack pointer or NIL.

L.SETFRAMNAM(NAM) - Set the frame name of the current frame to NAM. (Used by ERRORSET)

L.MKSTKP(CONTENTS oPOS) - Create a stack pointer to the unboxed stack pointer CONTENTS using OPOS. That is, if OPOS is a stack pointer, the frame it references, if any, will be released, and CONTENTS will be stored in OPOS. Otherwise a new stack pointer is created. Error if CONTENTS not an unboxed stack pointer.

L.RETTO(POS VAL) - Return to stack frame at POS with value VAL. POS may be an unboxed stack pointer or T meaning the topmost frame.

RELSTK(POS) - Release the stack frame (if any) referenced by the stack pointer POS. The contents of POS is set to "non-existent". If POS is not a stack pointer, nothing is done. Value is POS.

L.FLED1(POS) - If POS is a stack pointer, decrement the USE field of the referenced frame (if any), and set the contents of POS to 'non-existent". Similar to RELSTK but does not actually flush the frame. Used to implement the Interlisp functions RETTO, RETEVAL, ENVEVAL and ENVAPPLY.

L.ENVEVAL(FORM APOS CPOS AFLG CFLG) - Implements the Interlisp function ENVEVAL. Evaluates form with ALINK equal to APOS and CLINK equal to CPOS. APOS and CPOS may be stack pointers, literal atoms

meaning (STKPOS POS), integers meaning (STKNTH POS), T meaning top level frame, or NIL meaning the current frame. If APOS is a stack pointer and AFLG is T, APOS is released. Similarly if CPOS is a stack pointer and CFLG is T, CPOS is released.

L.ENVAPPLY (FN ARGS APOS CPOS AFLG CFLG) - Implements the Interlisp function ENVAPPLY. Applies FN to ARGS with ALINK equal to APOS and CLINK equal to CPOS. POS and FLG arguments same as for L.ENVEVAL.

GETTOPVAL(ATOM) - Returns the top most value of the literal atom. ATOM. If ATOM is not bound, returns NOBIND. Note that in a shallow bound environment this requires a stack search.

SETTOPVAL(ATOM VAL) - Set the top most value of the literal atom ATOM to VAL. If ATOM is not bound set the current value (=top level value) to VAL.

CLRSTK(FLG) - If FLG is NIL, releases all existing stack pointers. Otherwise returns a list of all existing stack pointers.

User Data Type Primitives

DEFTYPE(NWORDS NPTRS) - define a user data type containing NWORDS 16 bit words and NPTRS pointers. Value is the type number. Error if either argument is not an integer or is less than 0. Error if no more data types are available.

NALLOC (NTYP OLDONE) - Allocate a data type item of numeric type NTYP. If OLDONE is not NIL, its contents are copied into the newly allocated item. Error if NTYP not a legal, assigned user data type number.

TYPESTATUS(NTYP FLG) - If FLG is NIL, returns current status of user data type number NTYP. Status values are 0 - free type, 1 - in use, 2 - deallocated. If FLG is 1, set NTYP to be in use. If FLG is 2, set NTYP to be deallocated. Other values of FLG are ignored. In any case value is current status. Error if NTYP not a legal user data type number.

GETNPTRS(NTYP) - Returns the number of pointers in an item of user data type NTYP. Error if NTYP not a legal assigned user data type number.

GETNWRDS(NTYP) - Returns the number of 16 bit words in an item of user data type NTYP. Error if NTYP not a legal assigned user data type number.

General Structure Access Primitives

The following primitives are used by system level, implementation dependent, LISP functions to access components of data structures.

R16I(HEAD OFFSET) - Return, as a sign extended integer, the 16 bit quantity that is OFFSET 16 bit words from the beginning of the structure pointed to by HEAD.

R16S(HEAD OFFSET) - Return as an unboxed stack pointer the 16 bit quantity that is OFFSET 16 bit words from the beginning of the structure HEAD.

R16V(HEAD OFFSET) - Return, as a value cell pointer, the 16 bit quantity that is OFFSET 16 bit words from the beginning of the structure HEAD.

RPCAR(HEAD OFFSET) - Return the 24 bit CAR format pointer that begins OFFSET 16 bit words from the beginning of the structure HEAD.

RPCDR(HEAD OFFSET) - Return the 24 bit CDR format pointer that begins OFFSET 16 bit words from the beginning of the structure HEAD.

RNUM(HEAD OFFSET) - Return as an integer the 24 bit quantity that begins OFFSET 16 bit words from the beginning of the structure HEAD.

L.W16(HEAD OFFSET NEWVALUE) - Store the low order 16 bits of the pointer NEWVALUE in the location that is OFFSET 16 bit words from the beginning of the structure HEAD.

L.WCAR(HEAD OFFSET NEWVALUE) - Store the pointer NEWVALUE in CAR format in the locations beginning at OFFSET 16 bit words from the beginning of the structure HEAD.

L.WCDR(HEAD OFFSET NEWVALUE) - Store the pointer NEWVALUE in CDR format in the locations beginning at OFFSET 16 bit words from the beginning of the structure HEAD.

L.WNUM(HEAD OFFSET NUM) - Store the unboxed value of the integer NUM in the locations beginning at OFFSET 16 bit words from the beginning of the structure HEAD. Error if NUM is not an integer.

Miscellaneous Primitives

NTYP(PTR) - Return the integer data type of PTR.

EQP(X Y) - Return T if X and Y are EQ, equal integers, equal floating point numbers, or equal stack pointers.

CLOCK1() - Returns the value of a counter that is incremental once every 100 microseconds.

L.NCHR(X) - If X is a literal atom for a string, returns the number of characters in the pname or string. Returns NIL otherwise.

L.CHCON1(X) - If X is a literal atom or a string, returns the character code of the first character of X. Returns NIL if X is the empty string. Error if X not a literal atom or string.

L.SYSOUT(N) - Store the current virtual memory image on file number N.

L.SYSIN(N) - Run the virtual memory image stored on file number N.

L.LAPRD(X FN) - Load a compiled function. X may be an IO descriptor or string pointer. FN is the function name.

Stack Format

The stack is allocated in a fixed 64k word segment. This permits pointers to the stack on the stack to be 16 bits, thus reducing the size of stack frames and thereby increasing the speed of function calling.

The format of the frame extension is:

word 0	bit 15	0 extension is active 1 extension is inactive
word 0	bit 14	0 normal return 1 hard return - i.e. ALINK = CLINK, CXT > 0, or callers frame not contiguous
word 0	bit 13	garbage collector mark bit
word 0	bits 7:0	USE field
word 1	bits 15:0	the location of the last word -1 of the frame extension.
word 2	bits 15:0	BLINK, basic frame pointer
word 3	bits 15:0	ALINK
word 4	bits 15:0	CLINK
word 5	bits 13:8	High order 6 bits of LIT pointer (in interpreted functions LIT references the stack)
	bit 7	0 push the value upon return 1 discard the value
	bits 5:0	high order 6 bits of the return address
word 6	bits 15:0	low order 16 bits of return address
word 7	bits 15:0	low order 16 bits of literal pointer

32 bit temporary values

bits 15:8	0 value is a pointer 22 value is a 24 bit unboxed integer
bits 23:0	pointer or integer

The basic frame overhead is stored at the end of the basic frame so that the values of variables may be stacked as they are evaluated. The basic frame format is:

32 bit bindings

	bit 0	0 slot contains the current value of a variable
		1 slot contains the previous value of a variable(1)
	bits 21:0	value
word 0	bits 7:0	CXT field - the BLINK of the frame extension points here.
word 1	bits 15:0	pointer to the first word - 2 of the basic frame
word 2	bits 15:8	number of arguments in basic frame (unnecessary but convenient)
	bits 5:0	high order 6 bits of the frame (function) name
word 3	bits 15:0	low order 16 bits of the frame (function) name

Because it is possible to have a *fragmented* stack containing unused chunks, there is also a format for stack holes. The hole format is:

word 0	bits 15:0	all ones - this is the hole flag
word 1	bits 15:0	pointer to last word of hole
word 2	bits 15:0	pointer to previous hole, 1 if none
word 3	bits 15:0	pointer to next hole, 1 if none

When a frame is about to be run it is desirable to be able to detect quickly whether a hole immediately follows so the frame can be run without moving it. The unique hole flag permits this; neither a frame extension nor a basic frame can have all ones in the first word. The unique hole mark also permits easy merging of adjacent holes. The 2 way chain permits removal of a hole from the chain, and the chain itself permits reasonably quick searches for holes of adequate size. Note that we can end up with lonely 2 word holes that cannot be in the hole chain. These will only get used when

---

(1)
There is a bit for each binding to permit restart of the BIND instruction after a page fault.

required by the frame above or when they merge with adjacent holes.

Compiled Code Format

Compiled code is stored as a distinct data type. (In Interlisp-10 compiled code is stored in array space.) A compiled function is limited to 64K words and cannot cross a segment boundary in order to avoid the necessity for double precision addition to increment the program counter or to compute branch destinations.

Compiled code overhead is at the beginning of the code block and contains:

word 0	bits 15:0	total length of compiled code block
word 1	bits 15:0	location relative to word 0 of the beginning of the literals (the first FEF)
word 2	bits 15:8	number of FEF words in the literal area
word 2	bits 7:0	number of 16 bit value cell pointers in the literal area
word 3	bits 15:0	location relative to word 0 of the beginning of the 32 bit literals

In addition to the regular literals (constants) required by the compiled function, the literal area contains a lot of other information. The first thing in the literal area are the function entry frames, FEF's, for the function and for any internal PROGs and open LAMBDAs. An FEF contains the information required for binding the SPECVARS in the frame (and for unbinding in a context switch). The format of an FEF is:

word 0	bit 15:8	argument number of first specvar
word 0	bits 7:0	argument number of second specvar
word 1	bits 15:0	address of value cell for first specvar
word 2	bits 15:0	address of value cell for second specvar etc.

The first argument number is 1. The FEF is terminated by an argument number equal to 0.(1) Because of other constraints each FEF must be an even number of words.

---

(1) A bit mask to specify which arguments are specvars would use less space but would have made the microcode harder to write.

The FEF's are followed by some single word entries. First are 16 bit value cell pointers for specvars that are referenced (but not bound) in the function. Next are the constants required for referencing PVARS. PVARS are local vars in outer PROGs or the main function that are referenced from within a PROG that makes a frame. The left byte of the PVAR constant contains the relative location of the desired frame pointer in the current frame extension. (The PBIND operation gathers these frame pointers when a PROG is entered.) The right byte is the offset of the desired variable within the basic frame or frame extension.

Literal Atom Format

word Ø	bits 15:14	function type Ø regular LAMBDA 1 LAMBDA no spread 2 regular NLAMBDA 3 NLAMBDA no spread
	bits 13:7	number of arguments required (if compiled)
	bit 6	Ø expr. 1 compiled
	bits 5:Ø	high order 6 bits of compiled code or EXPR pointer
word 1	bits 15:Ø	low order 16 bits of code or EXPR pointer
word 2	bits 15:Ø	pointer to value cell - the high order 6 bits are constant; -1 means value cell does not exist.
word 3	bits 13:8	high order 6 bits of property list pointer
	bits 5:Ø	high order 6 bits of pname pointer
word 4	bits 15:Ø	low order 16 bits of pname pointer
word 5	bits 15:Ø	low order 16 bits of property list pointer

Value Cell Format

In order to save space in compiled functions, value cells are stored in a fixed 64K segment, so a value cell pointer can be 16 bits. Functions such as STKARGNAME and BACKTRACE need to know the variable names, so there must be a way to get from a value cell back to the atom. Thus a value cell contains the current value of the variable and a pointer back to the atom.

word 0	bits 13:8	high order 6 bits of litatom
	bits 5:0	high order 6 bits of value
word 1	bits 15:0	low order 16 bits of value
word 2	bits 15:0	low order 16 bits of atom

Function Cell Format

Since the contents of the function cell of an atom is more than a pointer, we have added a new data type, the function cell. A function cell is the value of GETD of a compiled function; and a function cell may be given as an argument to PUTD or may be used in APPLY, EVAL, etc. in the same manner as a litatom (the name of a function). The format of a function cell is the same as the format of the first two words of a litatom.

Pname Format

Pname characters are stored 2 8 bit characters per word in PDP-11 order, that is, the first byte is the low order byte. A pname begins on a word boundary and the first character is the length (in characters).

Array Format

In Interlisp-11, an array can contain only one type of entry. (Interlisp-10 allows arrays that contain both pointers and integers primarily so that compiled code can be stored in arrays.) The types of array are pointer, integer, floating point, and hash array. The size of an array is limited to 64K words, and arrays may not cross segment boundaries. The array overhead is:

word 0	bits 15:0	number of elements in array
word 1	bits 1:0	array type 0 pointer array 1 integer array 2 floating point array 3 hash array

The first three array types are 2 words per entry. Hash arrays use these words - each entry contains two 24 bit pointers.

Stack Pointer Format

Since the stack is limited to one fixed 64k segment, a stack pointer could be stored in 16 bits. However, to preserve the generality of

the general cons instruction, which expects 22 bits free list pointers, stack pointers are 32 bits long.

List Format

Lists are stored in a straightforward manner, two pointers per list cell. We did not implement any CDR coding because of lack of micro-code space.

word 0	bits 13:8	high order 6 bits of CDR
	bits 5:0	high order 6 bits of CAR
word 1	bits 15:0	low order 16 bits of CAR
word 1	bits 15:0	low order 16 bits of CDR

String Pointer Format

A string pointer contains the location of the first character of the string in string or pname character space and the length of the string. The format is:

word 0	bit 7	0 string begins at first (low order) byte of addressed word in string space
		1 string begins at second (high order) byte of addressed word in string space
	bits 5:0	high order 6 bits of pointer to characters
word 1	bits 15:0	low order 16 bits of pointer to characters
word 2	bits 15:0	length of string in characters

String Character Format

String characters are stored 2 8 bit characters per word, in PDP-11 order, i.e., the first byte is the low order byte.

Atom Hash Table Format

The atom hash table consists of a table of page addresses, and the pages referenced to allow expanding the hash table easily (expansion is not implemented yet). Each entry is 32 bits as follows:

word 0	bits 15:8	first character of the atom pname

	bits 5:0	high order 6 bits of the atom pointer
word 1	bits 15:0	low order 16 bits of the atom pointer

2 unique values (0 and 1, which are not legal pointers) are used to denote empty and reclaimed entries.

Integer Format

Integers are 24 bits wide stored in two words. The full 32 bits are not used to permit tagged unboxed integers to be stored on the stack.

Floating Point Format

Floating point numbers are 32 bits wide as follows:

word 0	bit 15	sign
	bits 14:7	exponent in excess 128 notation
	bits 6:0	high order 7 bits of fraction
word 1	bits 15:0	low order 16 bits of fraction

4. DIFFERENCES BETWEEN INTERLISP-11 AND INTERLISP-10

Atom hash table does not expand in Interlisp-11; does in Interlisp-10.

Interlisp-11 compiled code blocks and arrays are limited to 64K words each, no limit in Interlisp-10 (or about 100K words in fact).

Interlisp-11 strings are limited to 128K characters, Interlisp-10 strings are limited to 32K characters.

The maximum number of arguments for a function is 128 in Interlisp-11, 96 in Interlisp-10.

The maximum number of characters in an atom pname is 255 in Interlisp-11, 127 in Interlisp-10.

Arrays containing both unboxed integers and pointers are not allowed in Interlisp-11, are allowed in Interlisp-10.

Arrays containing 2 pointers per element do not exist in Interlisp-11. ELTD is implemented by creating a second array with a hash link from the first.

Control-H to interrupt at next function call does not exist in Interlisp-11, but control-B to interrupt immediately is (in principle) resumable without loss of any context.

In Interlisp-11 the garbage collector can be turned off, allowing storage to be allocated as needed.

PUTD(ATOM1 ATOM2) is like MOVD (ATOM2 ATOM1) in Interlisp-11. In Interlisp-10 ATOM2 becomes the (illegal) definition of ATOM1.

IO descriptors in Interlisp-11 allow much more flexible user control of I/O. IO descriptors do not exist in Interlisp-10.

In Interlisp-11 a function definition can be used anywhere a function name can be used in EVAL, APPLY, APPLY* etc. Not so in Interlisp-10.

The implementation of Interlisp-11 does not allow the function SETSTKARGNAME.

The Interlisp programming environment, i.e. break package, editor, history package, DWIM and CLISP, etc., has been incorporated into Interlisp-11. The only missing facilities are HELPSYS, which provides a user interface to the on-line manual, and BRKDWN, which provides a means for timing programs. These components contain a lot of machine dependent code and were not deemed essential at this time.

5. MEASUREMENT AND EVALUATION

The timings below compare the performance of Interlisp-11 with Interlisp-10 running on a lightly loaded KA10 processor with 256K words of memory. Time is in seconds.

	KA10			PDP-11/40E	
	cpu	elapsed	page faults	elapsed	page faults
dwimify	6	9	175	44	1089
(re)dwimify	.34	1	57	1	72
compile	9	14	?	12	217

Dwimify is an Interlisp function that translates from CLISP to regular LISP and also corrects errors. The (Re)dwimify example is dwimifying a function that has already been dwimified. The compile example is the Interlisp-11 compiler in both cases. From these figures we can conclude that Interlisp-11 is roughly comparable in CPU performance with Interlisp-10, but that it needs more than 128K words of memory. A page fault on the PDP-11 takes an average of 35 milliseconds, accounting for 38 of the 44 seconds in the dwimify example.

6. CONCLUSIONS

We have learned that in spite of the limitations of the hardware; i.e., the small number of general registers, 16 bit data paths and ALU, and the limited amount of microcode memory, a viable single user Interlisp machine can be and has been implemented. We believe that with the addition of another 128K words of memory, and even at current prices ($60K-$70K), Interlisp-11 provides a cost effective alternative to the large time sharing systems. A very small project at BBN spends $10K per year for computer time for one Interlisp programmer working half time.

We have been waiting for years for the "right" machine to appear. Since the requirements for LISP are not very different from say, Algol or Pascal, it seems to be a reasonable hope. The only commercially produced machines we know of that have remotely reasonable microprogramming, 24 bit or wider data paths and large virtual address space with small page size are the Prime 400 et al and the VAX 11/780. Both have some serious disadvantages.

The Prime 400 series was not designed for general emulation; and although a dispatch on any contiguous bit field is possible, it isn't always easy. An arbitrary shift requires as many as three micro-instructions. The microcode is neither clean nor elegant nor easy to use.

The VAX 11/780 is physically too large and too expensive ($160K for a minimal system) to be attractive as a single user LISP machine, but could make sense for a multiuser system. It has only 56 general registers and no means for generating constants. In addition, the user microcode is limited to either 1K of RAM or 3K of ROM, which leads to some rather difficult development problems.

We are considering both these machines for future versions of Interlisp. If we do transfer the system the most important issue to deal with is that of garbage collection. We also expect to gather more data about what should and should not be included in the instruction set. In particular, the next system will have enough microcode to allow the gathering of run time statistics. With more microcode, the next system can also be more machine independent. In addition, some of the internal structures, such as the atom hash table, should be changed to LISP structures of some sort. However, when discussing machine independence and transportability, it should be remembered that 16K of microcode is usually much harder to transfer than 16K of BCPL or machine language.

BIBLIOGRAPHY

[1] Ash, W. et al. Intelligent On-Line Assistant and Tutor System, BBN Report No. 3607, Bolt, Beranek and Newman, Cambridge, Mass., Jan. 1977.

[2] Baker, H.G. List Processing in Real Time on a Serial Computer, M.I.T. Artificial Intelligence Laboratory Working Paper 139, M.I.T., Cambridge, Mass., Feb., 1977.

[3] Bobrow, D.G. and Wegbreit, B. A Model and Stack Implementation for Multiple Environments, CACM, 16, 10, Oct. 1973.

[4] Bobrow, R. and Grignetti, M. Interlisp Performance Measurements, BBN Report No. 3331, Bolt Beranek and Newman, Cambridge, Mass., June, 1976.

[5] Clark, D.W. List Structure: Measurements, Algorithms and Encodings. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., August, 1976.

[6] Deutsch, L.P. A LISP Machine with Very Compact Programs. Third IJCAI, Stanford, California, 1973, pp. 697-703.

[7] Deutsch, L.P. and Bobrow, D.G. An Efficient, Incremental, Automatic Garbage Collector, CACM, 19, 9, Sept. 1976, pp. 522-526.

[8] Fuller, S. et al. PDP-11/40E Microprogramming Reference Manual, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., Jan. 1976.

[9] Greenblatt, R. The LISP Machine. M.I.T. Artificial Intelligence Laboratory Working Paper 79, M.I.T., Cambridge, Mass., Nov. 1974.

[10] Steele, G.L. Multiprocessing Compactifying Garbage Collection, CACM, 19, 9, Sept. 1975, pp. 495-508.

[11] Teitelman, W. et al. Interlisp Reference Manual, Xerox Palo Alto Research Center, Palo Alto, California, 3rd Revision, Oct. 1978.

[12] Urmi, J. A Machine Independent LISP Compiler and its Implications for Ideal Hardware. Ph.D. Dissertation, Department of Mathematics, Linkoping University, Linkoping, Sweden, 1978.

[13] Wadler, P.L. Analysis of an Algorithm for Real-Time Garbage Collection, CACM, 19, 9, Sept. 1976, pp. 491-500.

DISTRIBUTION

Director, Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209 Attn: Program Management	2
Scientific Officer Information Systems Program Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Mr. Gordon D. Goldstein, Code 437	3
Administrative Contracting Officer Defense Contracts Administration Services 660 Summer Street Boston, MA 02210	1
Director, Naval Research Laboratory Attn: Code 2627 Washington, D.C. 20375	6
Office of Naval Research Department of the Navy Arlington, VA 22217	6
Defense Documentation Center Bldg. 5, Cameron Station Alexandria, VA 22314	12
Office of Naval Research Branch Office 495 Summer Street Boston, MA 02210	1